Register Allocation
• When creating temporaries as you generate code, each temporary gets to sit in its own register
  • Temporaries are essentially “virtual” registers
• Real machines have a limited number of registers available for general purpose use
  • Called architectural registers — registers addressable by instructions (machines may have many more internal registers)
  • RISC-V: 32 integer registers and 32 floating point registers
    • But many of these registers are reserved for special purposes (stack pointer, frame pointer, return address, etc.)
    • In practice, fewer registers available
• What do you do with temporaries?
• Code generation uses a lot of temporaries; treat each temporary as a local variable that gets a spot on the stack
• Generate code for operations on temporaries the same way you generate code for operations on variables:
  • Load temporaries into registers
  • Perform operation
  • Store result in temporary
• How many registers does this need?
  • Why is this bad?
wouldn’t it be nice

- Other extreme: all temporaries are registers
  - No loads or stores required for temporaries
- All variables/local variables loaded into registers at the beginning of a function, saved back to memory at the end of the function
  - No “extra” loads and stores required for multiple uses of the same variable
- But this runs into the limit on the number of registers!
• One extreme doesn’t work (cannot keep all values in registers)
• The other extreme isn’t efficient (don’t want to keep loading/storing values)

• What if we pick some temporaries and variables to keep in registers?
  • Use registers for values we need, or need often
  • If we run out of space in registers, can spill registers to the stack (essentially, go back to treating it as a local variable)

• This is register allocation
Global vs. local

• Same distinction as global vs. local CSE

• Local register allocation is for a single basic block (BB)

• Global register allocation is for an entire function (but not interprocedural – why?)

• Will cover some local allocation strategies now, global allocation later
naïve register allocation

• For each basic block
  • Find the number of references of each variable
  • Assign registers to variables with the most references

• Details
  • Keep some registers free for operations on unassigned variables and spilling
  • Store **dirty** registers at the end of BB (i.e., registers which have variables assigned to them and whose value has changed)
  • Do not need to do this for temporaries (why?)
drawbacks

• Suppose we only have two “extra” registers for this code →

• What variables go into registers?

• Could we do better?

1: $T1 = A + B$
2: $T2 = A + T1$
3: $T3 = A + T2$
4: $D = A + T3$
5: $T4 = C + B$
6: $T5 = T4 + D$
7: $E = T5 + D$
Suppose we only have two “extra” registers for this code →

- What variables go into registers?
- Could we do better?

- Variables/temporaries that are **dead** do not need to be in registers anymore!
  - A and D can share a register
  - And so can all the temporaries!

```
1: T1 = A + B
2: T2 = A + T1
3: T3 = A + T2
4: D = A + T3
5: T4 = C + B
6: T5 = T4 + D
7: E = T5 + D
```
basic idea

• Perform register allocation on a *per basic block* basis
  • Register allocation across basic blocks is *global* — will discuss later
• Perform code generation and register allocation at the same time
  • Find registers for operands when translating 3AC to assembly
• Greedily reuse registers
  • Keep operands in registers if operand is *live* (later lecture on liveness)
  • If operand is already in register, no need for new loads
• Only store registers back to the stack if necessary
  • Need register for something else (*spill* register to stack/global memory)
  • At the end of basic block
tracking registers

• As code is generated keep track of:
  • What piece of data is in each register
    • In our case, two possibilities:
      1. Local variable/parameter or global variable
      2. Temporary
  • Whether the data is dirty (has its value changed since it was put into the register) — why?
1:  A = B + C  
2:  C = A + B  
3:  T1 = B + C  
4:  T2 = T1 + C  
5:  D = T2  
6:  E = A + B  
7:  B = E + D  
8:  A = C + D  
9:  T3 = A + B  
10: WRITE(T3)

1:  \{A, B\}  
2:  \{A, B, C\}  
3:  \{A, B, C, T1\}  
4:  \{A, B, C, T2\}  
5:  \{A, B, C, D\}  
6:  \{C, D, E\}  
7:  \{B, C, D\}  
8:  \{A, B\}  
9:  \{T3\}  
10:  \{\}
example

1: LW R1 B
   LW R2 C
   ADD R2 R1 R2
2: ADD R3 R2 R1
3: ADD R1 R1 R3
4: ADD R1 R1 R3
5: LW R1 R1
6: SW R3 C
1: A = B + C
2: C = A + B
3: T1 = B + C
4: T2 = T1 + C
5: D = T2
6: E = A + B
7: B = E + D
8: A = C + D
9: T3 = A + B
10: WRITE(T3)

\[
\begin{array}{|c|c|c|c|}
\hline
\text{Inst} & R1 & R2 & R3 \\
\hline
1 & B & A* & \\
2 & B & A* & C* \\
3 & T1* & A* & C* \\
4 & T2* & A* & C* \\
5 & D* & A* & C* \\
6 & D* & E* & \\
7 & D* & B* & \\
8 & A* & B* & \\
9 & T3* & & \\
10 & & & \\
\hline
\end{array}
\]
key operations

• **ensure**: make sure that a value exists in a register (put the value in the register, if necessary)
• **allocate**: find a register for a value (kick another value out of a register, if necessary)
• **free**: kick a value out of a register (save the value to the stack/global space if necessary)
key algorithms

For each tuple $C = A \text{ op } B$ in a BB, do

$R_x = \text{ensure}(A)$
$R_y = \text{ensure}(B)$
if $A$ dead after this tuple, $\text{free}(R_x)$
if $B$ dead after this tuple, $\text{free}(R_y)$
$R_z = \text{allocate}(C)$ //could use $R_x$ or $R_y$
generate code for op
mark $R_z$ dirty

At end of BB, for each dirty register
generate code to store register into appropriate variable

$\text{ensure}(\text{opr})$
if opr is already in register $r$
return $r$
else
$r = \text{allocate}(\text{opr})$
generate load from opr into $r$
return $r$

$\text{allocate}(\text{opr})$
if there is a free $r$
choose $r$
else
choose $r$ to free
free($r$)
mark $r$ associated with opr
return $r$

free($r$)
if $r$ is marked dirty and variable is live
generate store
mark $r$ as free
Some Details
what to free?

• Some flexibility in this algorithm: when kicking a value out of a register, how do you decide which register to free?

• Lots of choices, different implications!
  • Ideal: want to kick out the value that causes the fewest downstream spills (e.g., will be used the least in the future)
  • Some simple tie-breaking ideas:
    • First, prefer to kick out non-dirty values
    • Second, prefer to kick out values that are going to be used farthest in the future
global variables

• Algorithms presented assume that loading a value into a register can be done in one instruction
• Loading value from global variables may take multiple instructions
  • Compute/load address for global: LA t1, <address of x>
  • Load from address into register: LW t2, 0(t1)

• Some options to handle:
  • Always allocate an extra register for global variable’s address, only free it when value is freed (easy, but can waste registers)
  • Set aside register for address operations (even easier, but may require redundant address computations)
  • Treat loading address into register as another 3AC operation to process
Aliasing, as usual, is a problem

• What happens with this code?

//a and b are aliased
LD a R1
LD b R2
ADD R1 R2 R3
ST R3 c // c = a + b
R1 = 7 //a = 7
ADD R1 R2 R4
ST R4 d // d = a + b
Dealing with aliasing

• Immediately before loading a variable $x$
  • For each variable aliased to $x$ that is already in a dirty register, save it to memory (i.e., perform a store)
  • This ensures that we load the right value
• Immediately before writing to a register holding $x$
  • For each register associated with a variable aliased to $x$, mark it as invalid
  • So next time we use the variable, we will reload it
• Conservative approach: assume all variables are aliased (in other words, reload from memory on each read, store to memory on each write)
  • Better alias analysis can improve this
intersection

- Different optimizations interact with register allocation in different ways
  - Peephole optimizations can reduce register pressure, can make allocation better
  - CSE can actually increase register pressure (why?)
  - Different orders of optimization produce different results

- **Phase ordering** is an open problem in compilers
next: global register allocation